A Practical Tile Size Selection Model for Affine Loop Nests

Kumudha Narasimhan
Department of Computer Science and Automation
Indian Institute of Science
Bengaluru, Karnataka, India
kumudha@iisc.ac.in

Aravind Acharya[*]
NVIDIA
Redmond, WA, USA
aravinda@nvidia.com

Abhinav Baid
Birla Institute of Technology and Science
Pilani, Rajasthan, India
abhinavbaid@gmail.com

Uday Bondhugula
Department of Computer Science and Automation
Indian Institute of Science
Bengaluru, Karnataka, India
udayb@iisc.ac.in

ABSTRACT

Loop tiling for locality is an important transformation for general-purpose and domain-specific compilation as it allows programs to exploit the benefits of deep memory hierarchies. Most code generation tools with the infrastructure to perform automatic tiling of loop nests rely on auto-tuning to find good tile sizes. Tile size selection models proposed in the literature either fall back to modeling complex non-linear optimization problems or tackle a narrow class of inputs. Hence, a fast and generic tile size selection model is desirable for it to be adopted into compiler infrastructures like those of GCC, LLVM, or MLIR.

In this paper, we propose a new, fast and lightweight tile size selection model that considers temporal and spatial reuse along dimensions of a loop nest. For an n-dimensional loop nest, we determine the tile sizes by calculating the zeros of a polynomial in a single variable of degree at most n. Our tile size calculation model also accounts for vectorizability of the innermost dimension. We demonstrate the generality of our approach by selecting benchmarks from various domains: linear algebra kernels, digital signal processing (DSP) and image processing. We implement our tile size selection model in PolyMage (a domain-specific language and compiler for image processing pipelines) and Pluto (state-of-the-art polyhedral auto-parallelizer). Implementing the model in PolyMage allows us to extend it to DSP and linear algebra domains and also incorporate idiom recognition phases so that optimized vendor-specific library implementations could be utilized whenever profitable. Our experiments demonstrate a significant geomean performance gain of 2.2× over Matlab on benchmarks from the DSP domain. For PolyBench, we obtain a geomean speedup of 1.04× (maximum speedup of 1.3×) over Pluto.

[*]Affiliated with the Department of Computer Science and Automation, Indian Institute of Science during the time of this work.

CCS CONCEPTS

• **Software and its engineering → Compilers**.

KEYWORDS

Compilers, Loop tiling, Locality, Polyhedral compilation

ACM Reference Format:
Kumudha Narasimhan, Aravind Acharya, Abhinav Baid, and Uday Bondhugula. 2021. A Practical Tile Size Selection Model for Affine Loop Nests. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3447818.3462213

1 INTRODUCTION

Loops are often the performance bottlenecks for many applications and hence optimizing them is the key to obtaining higher performance on a given hardware. Loop tiling or loop blocking is one such important optimization used in general-purpose and domain specific compilers. Loop tiling uses data locality of the program to exploit the benefits of deep memory hierarchies. Tiling can be performed at multiple levels to fit the data in different levels of the memory hierarchy -

registers L1, L2 and LLC caches. The benefits of loop tiling can only be obtained when using correct tile sizes for these different levels.

Tile sizes for each dimension of a tiled loop nest have to be chosen carefully because (a) smaller tile sizes can lead to under utilization of the cache (b) larger tile sizes can lead to cache misses for computations within the tile or in some cases lead to insufficient work for all the cores. Although, many optimizing compilers (Pluto [4], PPCG [25]) and domain specific languages (DSLs) (LGen [16]) perform automatic tiling of loop nests they do not have a generic tile size selection model. They either rely on auto-tuning [34, 41] or model complex non-linear optimization problems [17, 26] to find good tile sizes which is time consuming and hence not suitable for a general purpose compiler.

In this paper, we propose a new fast and lightweight tile size selection model. Our intuition for the tile size selection model stems from the fact that tiling is profitable whenever there is reuse of data accessed in iterations of a loop nest. Therefore, having larger tile sizes along dimensions that exhibit reuse tend to utilize the cache efficiently. Our tile size selection model considers the temporal and spatial reuse along each dimension of a loop nest to calculate tile sizes for the corresponding dimensions. It also ensures that the memory footprint of a tile does not exceed the L1 cache capacity. Large tile sizes are chosen for vectorizable dimensions of the loop nest to obtain benefits of efficient vectorization and prefetching. Additionally, whenever problem sizes are small, smaller tile sizes are chosen to avoid load imbalance. Considering these factors, for an n-dimensional loop nest, we construct a polynomial which is of degree up to n: zeros of this polynomial are used to determine tile sizes for each dimension of the loop nest. This also allows programs with multiple loop nests to have different tile sizes for each loop depending on its relationship to access functions.

We implemented our tile size selection model in PolyMage - a domain specific language and compiler for image processing pipelines and Pluto - a state-of-the-art polyhedral auto-parallelizer. To demonstrate the generality of our approach, we extend PolyMage to digital signal processing and linear algebra domains. Using the PolyMage DSL, we not only prove the effectiveness of our tile size selection model but also incorporate an idiom recognition phase, so that optimized, vendor-specific library implementations can be utilized whenever profitable. Our experiments demonstrate that our implementation in PolyMage outperforms (i) state-of-the-art implementations that use vendor-specific libraries in the DSP domain (ii) polyhedral auto-parallelizers in cases where mapping to optimized library implementations is not profitable – due to efficient tile size selection model. Additionally, our model finds the tile sizes almost instantaneously without significant compile time overhead.

```
for (i = 0; i < N1; i++)
  for (j = 0; j < N2; j++)
    for (k = 0; k < N3; k++)
      for (l = 0; l < N3; l++)
        Op[i][j][k] += A[i][j][l] * C[l][k];
```

Figure 1: Multi-resolution analysis kernel.

Our contributions can be summarized as follows:

- We propose a generic and fast model for tile size selection in affine loop nests with arbitrary affine access functions.
- The model aims at effectively utilizing the cache while considering vectorization, prefetching and load balancing into account.
- We implement our tile size selection model in Pluto and PolyMage, and compare with the state-of-the-art frameworks. For DSP benchmarks, we obtain a geomean speedup of 2.2× over Matlab and for benchmarks from PolyBench, we obtain a geomean speedup of 1.04× (maximum speedup of 1.3×) over Pluto.
- The small compile time overhead of the model makes it suitable for adoption in a general-purpose compiler infrastructure, like MLIR.

The rest of the paper is organized as follows: Section 2 provides the necessary background on PolyMage and emphasizes the need for a tile selection model. We describe our tile size selection model in Section 3. Then Section 4 provides the implementation details in both PolyMage and Pluto. Section 5, presents our experimental methodology, demonstrating the impact of tile the proposed tile size selection model in both PolyMage and Pluto. Section 6 presents related work in this area and Section 7 concludes the paper.

2 MOTIVATION

In this section, we motivate the need for a good tile size selection model.

Tiling or blocking is performed on loop nests to exploit locality in computations. This helps to improve performance by reusing elements brought into cache from memory before being evicted. Further, a good tile size is one that is sized proportionately to fit the elements accessed in the computations perfectly within the cache. Choosing too small a tile size leaves cache space under utilized while too large a tile size may cause the computation to overflow the cache causing performance loss.

Consider the four level loop nest as shown in Figure 1. We tile the code and study the performance for different tile sizes (details of methodology in Section 5). We observe that choosing a good tile size can improve performance by 6.4× over running the same code with default tile size of 32 for each dimension of the loop. Therefore a good tile size selection model which computes the tile sizes quickly and efficiently is imperative to obtain the maximum performance from a given system.

Many optimizing compilers (Pluto [4], PPCG [25]) and domain specific languages (DSLs) (LGen [16]) perform automatic tiling of loop nests but do not have a generic tile size selection model. They either rely on auto tuning or choose a default tile size. Hence, a fast and generic tile size selection model is desirable for it to be adopted into compiler infrastructures like GCC, LLVM, or MLIR [15].

3 TILE SIZE SELECTION MODEL

This section describes the tile size selection model and illustrates its working with examples. The proposed tile size model is generic and works for any arbitrary affine access. The inputs to the model are (i) L1/L2 cache size, (ii) vector width and (iii) dimensional reuse along a dimension. We first describe the basic idea behind the model, then we present intra-tile optimzations that enable auto-vectorization and propose extensions to the model to reap the benefits of large vector widths and aggressive prefetchers.

3.1 Tile size calculation

At a high-level, the proposed tile size selection model assigns tile sizes to each dimension (loop) based on the amount of reuse available along that dimension. Firstly, we calculate dimensional reuse along each dimension. Secondly, we construct a reuse expression based on the memory access patterns in loop. This reuse expression is constructed for each loop and represents the total memory accessed by a tile. For a n-dimensional loop nest, this gives a k-degree polynomial where $k \leq n$. Then, by assuming the tile sizes to be proportional to dimensional reuse, we obtain a single variable, k-degree polynomial whose zeros give the tile sizes. The rest of this section explains each of these steps in detail using matrix-matrix multiplication as an example.

3.1.1 Calculating the tile volume. Tile volume (T) represents the total number of elements that can be stored in the cache for which we are tiling. The algorithm accepts L1/L2 cache size as an input in order to calculate the tile volume. Assuming an L1 cache size of 32KB and data size as the size of double i.e. 8 bytes, tile volume is 4096, i.e., 4096 distinct double values can fit inside the L1 cache.

3.1.2 Calculating dimensional reuse. We define dimensional reuse (γ) along a dimension as the number of temporal and group temporal reuse of the data along that dimension inspired by Wolfe et. al [37]. Consider a naive matrix-matrix multiplication with the standard three dimensional loop nest (i, j, k) for multiplying two matrices A and B to yield C.

Table 1: Reuse expressions.

memory access	no. of distinct accesses w.r.t tile size	no. of distinct accesses w.r.t dimensional reuse
$a[i]$	τ_i	$\gamma_i * \tau$
$a[\alpha * i]$	τ_i	$\gamma_i * \tau$
$a[i + j]$	$\tau_i + \tau_j$	$(\gamma_i + \gamma_j) * \tau$
$a[i - j]$	$\tau_i + \tau_j$	$(\gamma_i + \gamma_j) * \tau$
$a[i][j]$	$\tau_i * \tau_j$	$(\gamma_i * \gamma_j) * \tau^2$

The dimension i has a temporal reuse of 1 (owing to access $B[k][j]$), the dimensions j has a temporal reuse of 1 (owing to access $A[i][k]$) and k has a temporal reuse of 2 (considering both read and write operations for the access $C[i][j]$). We normalize these values and the dimensional reuse for each loop is:

$$\gamma_i = 0.5, \gamma_j = 0.5, \gamma_k = 1.$$

Note that, for a given dimension, its dimensional reuse is always a number.

3.1.3 Constructing the reuse expression. To effectively utilize the cache, tile sizes for all dimension should be selected such that the total number of distinct memory access in the tile should be equal to the tile volume (3.1.1). Assuming the tile size for dimension i to be τ_i, the number of distinct memory access for each array in the loop for a two dimensional loop is given in Table 1, Column 2. Considering the matrix-matrix multiplication example and assuming that the loop nests are tiled with tile sizes τ_i, τ_j and τ_k for loops i, j and k respectively. The memory accessed by each matrix (ref. Table 1, Row 4, Column 2), for this tile is:

$$\tau_i * \tau_j \text{ elements of matrix C,}$$
$$\tau_i * \tau_k \text{ elements of matrix A,}$$
$$\tau_k * \tau_j \text{ elements of matrix B.}$$

Equating these to the tile volume (T) we get:

$$\tau_i * \tau_j + \tau_j * \tau_k + \tau_k * \tau_i = T. \quad (1)$$

Note that Equation 1 cannot be constructed since we don't know the values of τ_i, τ_j and τ_k. However, the intuition for our tile size selection model is that the tile size for each dimension is proportional to its dimensional reuse, i.e., higher the dimensional reuse, the larger is the tile size along the dimension. Given γ_i and γ_j that represent dimensional reuse along i and j dimensions of a loop nest respectively, we have $(\gamma_i / \tau_i) = (\gamma_j / \tau_j)$ = constant, say τ. If this constant is different for each dimension, one cannot have tile sizes proportional to dimensional reuse. Thus, the tile size along dimension i can be written with $\tau_i = \gamma_i * \tau$, $\tau_j = \gamma_j * \tau$ and $\tau_k = \gamma_k * \tau$.

Therefore, Equation 1 can be re-written as:

$$(\gamma_i * \gamma_j + \gamma_j * \gamma_k + \gamma_k * \gamma_i) * \tau^2 = T. \quad (2)$$

```
for (int i = 0; i <= t1; i = (i + 1))
  for (int j = 0; j <= t2; j = (j + 1))
    ybs[i] += yds[(M + i) - j] * window[j];
```

Figure 2: Generated DSP code from vuvuzela filter.

Note that Equation 2 is a single variable polynomial and can then be solved for τ and tile sizes can be computed. Substituting dimensional for matrix multiplication in the above equation we get:

$$(0.5 * 0.5) * \tau^2 + (0.5 * 1.0) * \tau + (1.0 * 0.5) * \tau = 4096.$$
$$\implies 1.25 * \tau^2 - 4096 = 0.$$

Solving the above equation yields the roots +57.24 and −57.24. Discarding the negative one, the tile size for the loops i, j and k can be obtained by multiplying the floor of the positive root with its corresponding dimension reuse. Thus, the final tile sizes for the matrix-matrix multiplication code is $\tau_i = 28, \tau_j = 28$ *and* $\tau_k = 57$.

This model provides good tile sizes since the reuse along a dimension also represents the number of data points that need to be received from the previous tile along that dimension. Thus, higher reuse implies, higher data movement. Therefore, increasing the tile size along that dimension, reduces the number of tiles which in turn reduces the total communication across tiles in the computation.

An interesting case in Table 1 with accesses of the form $a[i - j]$, where the number of distinct memory access is still the sum of the tile sizes along the dimensions i and j. This is explained with an example code from the DSP domain shown in Figure 2. Let the tile size for dimensions i and j be $\tau 1$ and $\tau 2$ respectively. Then the amount of memory accessed by the tiled loops i and j by array ybs is $\tau 1$ and by array $window$ is $\tau 2$. The memory required for yds is calculated as $(\tau 1 - 0) - (0 - \tau 2)$ which is equal to $\tau 1 + \tau 2$. The reuse expression can be constructed as $(\tau 1) + (\tau 1 + \tau 2) + (\tau 2) = T$, which can be further reduced to a single variable polynomial and solved to obtain the tile sizes.

Algorithm 1 describes our tile size selection model for a permutable band [1] G. For each dimension d in G, the algorithm computes the dimensional reuse (line 1) along d as the sum of temporal and group temporal reuse. The memory accesses in G is collected in line 3. The reuse expression for G can be obtained by computing the tile volume, using the number of distinct memory access in a tile and expressing the tile size of each dimension as a factor of dimensional reuse. Further, the algorithm computes the reuse expression (line 4) and the positive root of Equation 2 determines the tile sizes (lines 5-7). It is desirable that the innermost dimension have a large tile size, when it is vectorizable. Therefore, the tile size corresponding to the innermost dimension (innerDim) is set to the smallest of the bound on the innermost dimension (obtained in Line 2), or innerTileSize (an input to the algorithm). The algorithm to find the innermost dimension of a group is described subsequently in Section 3.2.

[1] Permutable band is a set of consecutive loops which can be permuted.

Algorithm 1: ComputeTileSize

```
Input: Permutable band $G$, cacheSize, innerTileSize,
       innerDim, nDims
Output: Tile sizes for each dimension of $G$
1 dimReuse[1...nDims] ← getDimReuse($G$)
2 innerDimSize ← getInnerDimSize($G$)
3 memAccess ← get memory references in $G$
4 reuseEqn ← getReuseEquation(memAccess, dimReuse,
    innerDim, tileSize)
5 $\tau \leftarrow \lfloor$positiveRoot(reuseEqn)$\rfloor$
6 foreach $i \in \{1, 2, \ldots, nDims\}$ do
7     tileSizes [i] ← dimReuse[i] × $\tau$
8 tileSizes[innerDim] ← min(innerDimSize, innerTileSize)
9 return tileSizes
```

3.2 Intra-tile Optimization

Exploiting SIMD parallelism can lead to a good improvement in performance. PolyMage relies on host compilers like gcc/icc for auto-vectorization. These compilers typically generate efficient vector code when: (i) the innermost dimension is parallel, and (ii) the innermost dimension has stride-0 or stride-1 accesses.

Algorithm 2 tries to determine a "vectorization friendly" dimension to be at the innermost level for each group. Assuming each dimension d in a permutable band of loops G is the innermost dimension, the algorithm finds, per iteration of the innermost dimension, the number of memory accesses which have spatial reuse (s), temporal reuse (t), and the total number of distinct memory accesses (a) (lines 2-4).

Algorithm 2: IntraTileOptimize

```
Input: A permutable band $G$
Output: Dimension to be made the innermost in $G$
1 foreach $d \in G$ do
2     s ← getNumSpatialReuse($d$, $G$)
3     t ← getNumTemporalReuse($d$, $G$)
4     a ← getTotalAccess($d$, $G$)
5     v ← isVectorizable($d$, $G$)
6     score[d] ← score[d] +
          $(2*s)+(4*t)+(8*v)-(16*(a-s-t))$
7 innerDim ← getDimWithMaxScore(score)
8 return innerDim
```

Table 2: Intra-tile optimization score for matmul.

dim	s	t	v	a	score
i	0	1	false	4	-44
j	3	1	true	4	18
k	1	2	false	4	-6

A dimension is considered vectorizable (v=1) if it is parallel and does not result in non-contiguous accesses in the innermost iteration (line 5). The score for the dimension is calculated using the heuristic in line 6 which favors a parallel dimension that has stride-0 access (temporal reuse), stride-1 access (spatial reuse) and smaller number of accesses with high scatter-gather distances. The intra-tile iterator corresponding to the dimension with the highest score and is permuted to the innermost level of the loop nest. This loop is marked vectorizable if it is parallel.

For example, consider a naive matrix-matrix multiplication with three dimensional loop nest. The scores for each dimension (loop) is shown in the Table 2. The loop k carries a dependence while the loop i, when permuted to the innermost level, results in non contiguous accesses for arrays C and A. Hence both i and k are not considered vectorizable. The algorithm finds the highest score for the j loop as it is parallel and has stride-0 and stride-1 accesses only. The intra-tile iterator corresponding to loop j is permuted to the innermost level.

The correctness of Algorithm 2 follows from the fact that, a permutable band represents a set loops that can be tiled. Hence all the intra and inter-tile iterators can be permuted. Algorithm 2 picks an intra-tile loop and then moves it to the innermost level and none of the dependences are violated.

Stock et. al [32] proposed a model to choose intra-tile loop order that maximizes stride-0 and stride-1 accesses by solving an ILP problem. However, we observe that in practice, Algorithm 2 achieves the same objective without solving ILPs, making it more suitable for adoption in a general-purpose compiler infrastructure.

3.2.1 Improvements to the tile size selection model. In this subsection, we describe the improvements to the tile size selection model described in Section 3.1. Smaller tile sizes along the vector dimension can diminish the benefits of vectorization and prefetching. Hence, we modify the tile size selection model to fix the tile size of dimension corresponding to the vectorizable loop to 256. For example, in the case of matrix multiplication, the inner-most loop after intra-tile optimization is the j loop. Fixing $\tau_j = 256$ and re-writing the Equation 2 as:

$$(0.5 * \tau) * 256 + 256 * (1.0 * \tau) + (1.0 * 0.5) * \tau^2 = (32768/8).$$

$$\implies 0.5 * \tau^2 + 384 * \tau - 4096 = 0.$$

Solving the above equation yields the roots 10.52 and −778.52. Discarding the negative one, the tile size for the loops i, j and k can be obtained by multiplying the floor of the positive root with its corresponding dimension reuse. The final tile sizes obtained are $\tau_i = 5, \tau_j = 256$ *and* $\tau_k = 10$.

3.3 Tiling and Parallelism

Tile sizes can have an effect on the amount of parallelism and the amount of work for each thread. Assigning a large tile size to the outer-parallel dimension when there are fewer number of iterations along that dimension leads to very few tiles. This causes load imbalance leading to under utilization, affecting performance negatively. Hence, if the loop bounds/ iteration space bounds are available at the time of tile size calculation, we can use this information in the tile size selection model and reduce load imbalance. Let us consider a matrix-matrix multiplication example where the loop bounds for i, j, k are known to be 64, 64, 64. We can calculate the total size needed for the computation to be $3 * 64 * 64$ for the three matrices. Given the number of cores in the system as 16, the tile volume (T) used in Equation 2 can be calculated as

$$T = total\_computation\_size/num\_cores, \quad (3)$$

which is $3 * 64 * 64/16 = 768$. Reducing the tile volume from 4096 to 768 ensures smaller tile sizes and avoids load imbalance.

To summarize, our tile size selection model

- is generic and handles arbitrary affine accesses,
- finds tile sizes from zeros of a single variable polynomial. This is significantly faster when compared to auto-tuning or solving an optimization problem,
- considers efficient prefetching and vectorization,
- accounts for load balancing when problem sizes are known at compile time, and
- can further be extended to multi-level tiling by accepting multiple cache sizes with minor modifications to Algorithm 1.

4 IMPLEMENTATION DETAILS

In this section, we describe the extensions that were made to PolyMage [24] and Pluto [22] to implement the tile size selection model.

4.1 Enhancements to PolyMage

Figure 3 shows the modifications to the stages in the PolyMage compiler. First, we extend PolyMage's specification to include language constructs that declare matrices and

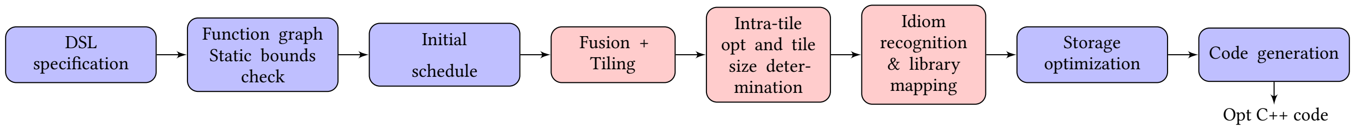
DSL specification
Function graph Static bounds check
Initial schedule
Fusion + Tiling
Intra-tile opt and tile size determination
Idiom recognition & library mapping
Storage optimization
Code generation
Opt C++ code

Figure 3: Modified compiler flow in PolyMage (Red denotes modified stages).

overload operators for basic matrix operations. The list of matrix operators and functions supported in PolyMage are provided in Tables 9 and 10 in the supplementary material. We also extend the fusion and tiling stage in the PolyMage compiler to handle reduction operations. We introduce a new intra-tile optimization pass to bring parallel loops to the innermost level in order to enable auto-vectorization (described in Section 3.2). The tile sizes for this new schedule is found using the approach presented in Section 3. We then introduce a new idiom recognition stage that maps basic matrix operations to optimized library calls. After this step, the storage optimizations described in Section 2 are performed and optimized C++ code is generated. The idiom recognition stage or mapping to function call is described in detail below.

Idiom recognition stage maps (i) basic matrix operations to highly tuned BLAS implementations (OpenBLAS or Intel MKL) only if $M * N * K \geq (256)^3$ to ensure such a mapping to a library call is profitable [2], where M, N and K are the upper bounds of the loops representing the computation(ii) Fourier transforms are always mapped to FFTW subroutine calls, as they reduce the complexity of computing discrete Fourier transform from $O(N^2)$ to $O(N \log N)$.

4.2 Enhancements to Pluto

In this section, we provide the details on Pluto's implementation of the tile size selection model proposed in Section 3.1. Pluto is the state-of-the-art tool to optimize affine programs for multicore CPUs using polyhedral transformation techniques. Figure 4 shows the Pluto compiler flow and the box in red represents the newly added stage. The compiler accepts unoptimized C code and extracts the polyhedral representation of the program. Pluto finds affine loop transformations that maximize locality and parallelism, by finding a permutable band of loops that can be tiled. It then performs loop tiling and the tile sizes are determined using the model described in Section 3. It further performs intra-tile optimization similar to the one explained in Section 3.2. Lastly it performs unroll-and-jam optimization on ClooG's AST and generates OpenMP parallelized C code.

Unroll-and-jam optimizations provide good register reuse and the benefits of tiling with small tile sizes. Kong et. al [14], demonstrated that with unroll jam, performance on par with tuned tile sizes can be obtained without tiling loop nests. This indicates that unroll-jam optimizations can be used to efficiently exploit reuse in both registers and L1 cache. With unroll-and-jam optimizations enabled in Pluto, we aim to exploit reuse at L2 cache instead of L1. Hence, we use L2 cache size as a parameter to Algorithm 1 instead of L1 cache size. We also approximate the tile sizes found by the model to the nearest multiple of the unroll jam factor to ensure the profitability of unroll-and-jam optimizations in full tiles. Further, we also choose to fix the tile size of the vector dimension to 512 instead of 256.

Tiling for L2 cache forces the model to choose larger tile sizes. This could lead to load imbalance when the number of tiles in the phase that has parallelism is small. If the loop bounds are known at compile time, then an approach similar to the one described in Section 3.3 may be adopted in Pluto, ensuring load balance. For loop nests with parametric bounds, we assume a default tile size of 32 along the parallel wavefront, while the tile sizes for the remaining dimensions are found using the proposed tile size selection model.

5 EMPIRICAL RESULTS

In this section, we describe the details of our experiments. As discussed in the previous section, we implement our tile size selection model in PolyMage and Pluto. Extending PolyMage to linear algebra and digital signal processing domains allows us to evaluate the generality of the tile size selection model in presence of various optimized vendor specific library implementations whenever possible. We also evaluate our work against state-of-the-art polyhedral auto-parallelizers. We realize our tile size selection model in two polyhedral auto-parallelizers i) PolyMage referred as *PolyMage-opt*, and ii) Pluto [22], referred as *Pluto-tss*, in the rest of this section.

5.1 Experimental Setup

All our experiments are run on a dual socket, NUMA based multicore system with Intel Xeon processors based on Skylake-SP microarchitecture. We use Intel C/C++ compiler (version 19.1) to compile optimized C++ codes generated by PolyMage and Pluto. OpenMP threads were explicitly pinned to cores 0 to 15 and hyperthreading was disabled. Matrix operations are mapped to BLAS implementations provided by Intel's Math Kernel Library (MKL, version 19.1.2) [20]. The details

[2]This value was obtained experimentally on our setup by measuring BLAS performance while sweeping the design space. This value also correlates with prior observations in [11].

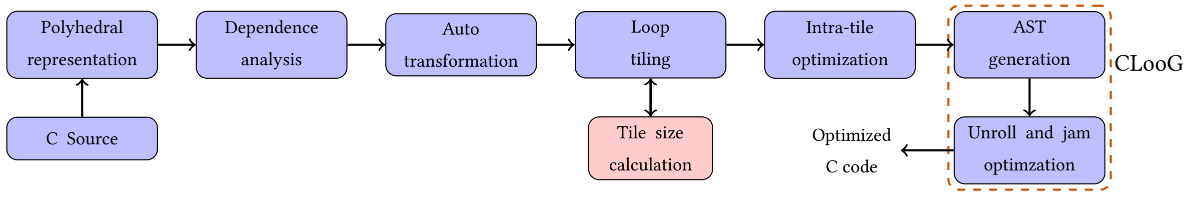

Figure 4: Modified compiler flow in Pluto (Red denotes the newly added stage).

of our experimental setup is provided in Table 3. We take hundred runs for each of the benchmarks. We discard the first two runs and report the average of the others as the execution time of the benchmark.

Table 3: Experimental setup.

Processors	Intel(R) Xeon(R) Silver 4110 CPU
Clock	2.10 GHz
Cores	16 (8 per socket)
Hyperthreading	disabled
Private caches	32 KB L1 cache, 1 MB L2 cache
Shared cache	11 MB L3 cache
Memory	256 GB DDR4
Matlab version	9.9.0.1524771 (R2020b)
Scipy version	1.0.0
Compiler	Intel C/C++ (icc/icpc) 19.1.2.254
Compiler flags	-O3 -xhost -qopenmp -fma -ipo
OS	Ubuntu 18.04.5 LTS

5.2 Benchmarks

We evaluate our approach on benchmarks from a wide range of domains consisting of digital signal processing (2 filters), and image processing benchmarks (6 benchmarks from PolyMage [21]) and benchmarks from PolyBench [23] suite. The PolyBench suite has been widely used to evaluate the performance of polyhedral compilers. We evaluate the entire PolyBench suite with *Pluto-tss*. Some benchmarks like lu, ludcmp and cholesky fall outside the class of programs that can be represented in PolyMage and hence PolyMage evaluates 13 linear algebra benchmarks from PolyBench. We use Pluto (version 0.11.4-955) [22] and PPCG (version 0.08.2) [25] as the baselines for our comparison. No additional flags were used with Pluto as it performs tiling, OpenMP parallelization, intra-tile optimizations and unroll-jam of loop nests (with an unroll-jam factor of 8) by default. We generated tiled, OpenMP-parallelized, C code with PPCG using the flags `--target=c`, `--openmp` and `--tile`. Codes generated by PPCG failed to compile with the Intel icc/icpc compilers for symm, trmm, correlation, gramschmidt and head-3d. For these benchmarks, we used GCC v8.0 and mapped BLAS calls to OpenBLAS library v0.2.20 with PolyMage. Loop nests were tiled with the default tile sizes (32 for Pluto and 16 for PPCG). We use both *extra-large* and *medium* datasets for evaluation. We evaluate *Pluto-tss* on the extra-large dataset alone, as the problem sizes in medium are very small to ensure load balancing and would require the problem sizes to be known for the model ensure load balancing while efficiently utilizing the cache, for every benchmark.

We consider the *unwanted spectral* and *vuvuzela* filter benchmarks from the DSP domain. The *unwanted spectral* filter uses a low pass filter to remove noise in the input signal. The *vuvuzela filter* is used to filter out the Vuvuzela noise from the input signal. These filters have reduction operators with arbitrary affine accesses in their loop nests as shown in Figure 2. Domain experts rely on hand optimized implementations from vendors like Matlab (with parallel computing support) and Intel's Scipy for high performant implementations, which serve as the baselines for our comparison with PolyMage-opt. We map all the upsampling operations in these filters to routines in the `fftw3` library.

For image processing applications, Jangda et al. [13] propose several optimizations, including a dynamic programming based tile size selection model, which are implemented in PolyMage. Hence for applications in image processing domain their approach serves as a strong baseline for comparison with the tile size selection model in our approach.

5.3 Performance Analysis

The objective of this section is to analyze the benefits of optimizations discussed in Section 3 on performance from various domains written. We compare with state-of-the-art approaches/libraries in respective domains.

5.3.1 PolyBench Benchmarks. The execution times for linear algebra benchmarks from PolyBench suite with baseline polyhedral auto-parallelizers and PolyMage-opt are listed in Tables 4 and 5 for medium and extra-large datasets respectively. *PolyMage* represents the baseline configuration without BLAS-mapping and using a default tile size of 32 along each dimension. *PolyMage-tss* represents the configuration in PolyMage with the tile size model described in the paper. The tables present the execution times for 1 thread and 16 threads for each baseline. Note that Pluto and PPCG

Table 4: Execution and tile size selection time (ms) with *medium* dataset for linear algebra benchmarks from PolyBench.

Benchmark	Pluto		PPCG		PolyMage		PolyMage-opt		PolyMage-opt speedup over			Tile size selection
	1	16	1	16	1	16	1	16	Pluto	PPCG	PolyMage	time (ms)
2mm	2.7656	0.7068	13.914	3.1354	4.4246	1.1939	2.6816	0.4328	1.63	7.24	2.76	10.9
3mm	4.9388	1.2082	20.882	4.57	6.5696	1.6698	4.3086	0.5902	2.05	7.74	2.83	13.6
atax	0.3828	0.1432	0.6516	0.1204	0.1406	0.0719	0.1380	0.0718	1.99	1.68	1.00	4.31
bicg	0.3812	0.1324	0.9316	0.1112	0.1396	0.0942	0.1390	0.0943	1.40	1.18	1.00	4.44
doitgen	2.6632	22.763	7.7126	15.8	2.3316	2.442	1.6710	0.3784	60.16	41.75	6.45	7.15
gemm	1.8878	0.4476	10.748	2.1384	2.8243	0.5736	2.1764	0.2466	1.82	8.67	2.33	5.02
gemver	0.7204	0.181	0.7528	0.1652	0.3283	0.2085	0.3120	0.2080	0.87	0.79	1.00	5.81
gesummv	0.1746	0.0548	0.2576	0.0508	0.0553	0.026	0.0553	0.0265	2.07	1.92	0.98	4.43
mvt	0.8984	0.1246	0.6702	0.1022	0.1166	0.1016	0.1166	0.1018	1.22	1.00	1.00	3.76
symm	8.0828	11.461	10.872	8.085	3.3313	2.9656	2.5854	1.7080	6.71	4.73	1.74	10.1
syr2k	7.184	2.1842	7.4706	2.2406	7.297	2.0539	5.0828	1.8458	1.18	1.21	1.11	6.97
syrk	4.2988	1.1994	5.6366	1.623	3.8873	1.0934	3.6700	1.3614	0.88	1.19	0.80	6.68
trmm	1.0144	0.8734	5.0386	1.1368	1.5556	0.5065	1.2540	0.3420	2.55	3.32	1.48	4.82
geomean									**2.26**	**2.96**	**1.53**	

Table 5: Execution time (s) with *extralarge* dataset for linear algebra benchmarks from PolyBench.

											Speedup (16 threads)			
Bench-	Pluto		PPCG		PolyMage		PolyMage-tss		PolyMage-opt		PolyMage-opt over			PolyMage-tss
mark	1	16	1	16	1	16	1	16	1	16	Pluto	PPCG	PolyMage	PolyMage
2mm	3.294	0.311	14.44	1.456	2.912	0.315	3.688	0.323	0.964	0.089	3.49	16.35	3.54	0.98
3mm	4.801	0.435	21.08	2.041	6.623	0.733	6.188	0.552	1.569	0.143	3.03	14.23	5.11	1.33
atax	0.019	0.003	0.024	0.003	0.007	0.001	0.007	0.001	0.007	0.001	1.92	2.14	1.04	1.08
bicg	0.020	0.003	0.022	0.003	0.007	0.001	0.007	0.001	0.007	0.001	1.82	1.82	0.98	1.00
doitgen	0.911	0.919	3.632	1.098	1.277	0.252	0.778	0.069	0.866	0.078	11.75	14.03	3.22	3.65
gemm	2.802	0.205	12.96	0.999	3.389	0.286	3.296	0.275	0.868	0.078	2.62	12.78	3.66	1.04
gemver	0.120	0.015	0.193	0.022	0.088	0.012	0.063	0.010	0.063	0.010	1.43	2.10	1.12	1.10
gesummv	0.048	0.005	0.054	0.005	0.019	0.002	0.017	0.002	0.017	0.002	2.93	2.99	1.23	1.24
mvt	0.142	0.014	0.110	0.012	0.033	0.007	0.034	0.007	0.034	0.007	1.85	1.65	0.99	1.01
symm	55.56	55.88	20.29	15.01	6.281	3.890	4.071	2.441	4.084	2.432	22.98	6.17	1.60	1.59
syr2k	9.060	1.255	10.45	1.497	8.887	1.333	6.656	1.021	6.656	1.005	1.25	1.49	1.33	1.30
syrk	5.029	0.693	6.923	0.939	4.518	0.638	4.188	0.631	4.197	0.628	1.10	1.49	1.02	1.01
trmm	1.392	0.155	5.404	0.486	1.685	0.170	1.912	0.166	1.920	0.167	0.93	2.92	1.02	1.03
geomean											**2.61**	**4.00**	**1.65**	**1.24**

use default tile size of 32 and 16 respectively. PolyMage-opt for medium dataset does not invoke routines from BLAS libraries. Hence the configuration is the same as *PolyMage-tss*. Even though both Pluto and PPCG perform the similar optimizations as *PolyMage-tss*, both of them lack a tile size selection model. Apart from good tile sizes, when the loop bounds for a particular dimension are very small, PolyMage-opt chooses not to tile the loop nest, there by avoiding loop tiling overhead completely. Therefore, with Polymage-opt, we observe an improvement of 2.26×, 2.96× over PPCG and 1.53× over PolyMage for medium datasets with 16 threads (Columns 10-12 in Table 4). The tile size selection model is the only contributor for the performance improvement of PolyMage-opt over PolyMage. In the case of gemver we observe a slowdown of 13% over Pluto for 16 threads. This is because (i) Pluto finds a different loop permutation that enables fusion (ii) Pluto performs unroll-jamming to further improve register reuse.

For symm and doitgen, spurious loop carried WAR dependences on temporary variables prevents Pluto from tiling and parallelizing the loop nests. However, in our approach these dependences do not exist as these temporary variables inside a loop nest are represented using arrays whose dimensionality is equal to the nesting depth of the statement containing the definition of the scalar variable. Therefore, we see significant performance improvement of 60× and 6×, for doitgen and symm respectively, over Pluto for medium dataset.

Comparison of PolyMage-opt with PolyMage for doitgen benchmark reveals the impact of the tile size selection model. In case of medium datasets (Table 4), default tile size of 32 will result in small number of working threads leading, to load imbalance, which is evident from the comparison of single-thread and 16 thread execution times of PolyMage. By choosing a smaller tile size, PolyMage-opt ensures load balance in smaller benchmarks.

Our heuristic described in Section 4.1 maps matrix-matrix multiplications for large and extralarge sizes to Intel MKL's implementation of BLAS. Hence in case of extralarge datasets with 16 threads, PolyMage-opt provides a geomean improvement of 2.61× over Pluto, 4× over PPCG and 1.65× over PolyMage (Columns 12-14 Table 5). The ability of our approach to utilize optimized BLAS routines when the size of the matrices are large, or perform polyhedral optimizations for smaller problem sizes allows us to incorporate the best of both worlds. Further, we observe that that even without BLAS mapping, *PolyMage-tss* provides a performance benefit of 1.96×, 3× and 1.24× over Pluto, PPCG and PolyMage respectively (last column of Table 5).

Table 6 lists the execution time of all the PolyBench benchmarks for Pluto and PPCG with default tile size and *Pluto-tss*, which implements the tile size selection model for single thread and 16-thread executions and respective speedups are provided in Columns 8-12. The last column represents the time taken by the tile size selection model in Pluto. We observe that *Pluto-tss* provides a geomean speedup of 1.04× and 5.11× over Pluto and PPCG respectively. We provide the scaling results for *Pluto-tss* and *PolyMage-tss* in the supplementary material (Tables 11 and 12). Note that, the impact of the tile size selection model is less pronounced in Pluto for the following reasons: 1) the base line version of Pluto implements unroll-and-jam optimizations, which diminish the impact of loop tiling in general, as described in Section 4.2, and 2) parametric loop bounds in PolyBench benchmarks forces implementation in Pluto to choose fixed tile sizes along the parallel wavefront to avoid load imbalance. If the parameter values are known, they can be used to effectively compute the tile volume while ensuring load balance, as described in Section 3.3. We observe that the tile size selection model provides a maximum speedup of over 2× in the single threaded case and over 1.3× on 16 threads, indicating that *Pluto-tss* efficiently captures locality but falls short in the parallelism aspect. During our experiments, we observed that tile sizes along the parallel wavefront to 32, has the following impact on performance: 1) performance loss on benchmarks like gemm, 2mm, 3mm, syrk and syr2k where larger tile sizes are preferred, 2) ensures better load balance and hence results in performance gains on benchmarks like lu, cholesky and trisolv, where wavefront parallelism is exploited. Tile sizes that efficiently utilize cache while ensuring load balancing can be ensured only if the loop bounds are known at compile time, or if parametric loop tiling infrastructure is available, then the tile sizes can be determined at runtime.

5.3.2 DSP Filters Benchmarks. Table 7 provides the execution times for vuvuzela and unwanted spectral filters. *PolyMage-fft* represents the configuration where upsampling operations are matted to fftw library calls but all other loops are tiled with default size of 32. Since, these filters involve significant number of reductions, just optimizing upsampling operations with fftw library does not scale with the increase in number of cores. PolyMage-opt has the ability to optimize reduction operations by tiling using the tile size model proposed. We compare our implementation with Intel's SciPy and Matlab with parallel computing tool box and show that for 16 threads, PolyMage-opt provides a mean speedup of 11.8× over SciPy and 2.2× over Matlab. We also observe that, among the implementations of the filters, Intel's SciPy provides best sequential (1 thread) performance. However, as shown in Table 7, using Intel's SciPy does not scale well with the number of cores. Further, when compared to PolyMage-fft (which maps to fftw3 but uses default tile sizes for other loops), we see a performance improvement of 15%. Overall, our approach PolyMage-opt provides better performance than state-of-the-art, demonstrating the effectiveness of compiler based optimizations in conjunction with optimized library routines.

5.3.3 Image Processing Benchmarks: To recall, the baseline PolyMage's tile size selection for image processing applications is based on a dynamic programming model [13]. Since both PolyMage and PolyMage-opt perform same optimizations on these benchmarks, the objective here is to compare the tile size selection routines. We observe that in most of the cases PolyMage-opt finds the same tile sizes as PolyMage. In cases where different tile sizes were found, the difference of tile sizes (per dimension) very small (±2). Therefore, for image processing benchmarks (Unsharp Mask, Harris Corner, Bilateral Grid, Multiscale Interpolate, Camera pipeline and pyramid blend) PolyMage-opt performs within ±4% of baseline, which is within experimental bounds. Thus, the proposed tile size selection model does not introduce any performance regression for image processing pipelines.

Table 6: Execution and tile size selection time (s) with *extralarge* dataset for benchmarks from PolyBench using Pluto.

Benchmark	Execution time (s)						Pluto-tss speedup over				Tile size selection
	PPCG		Pluto		Pluto-tss		PPCG		Pluto		
	1	16	1	16	1	16	1	16	1	16	time (ms)
2mm	14.271	1.448	3.316	0.296	2.556	0.255	5.58	5.69	1.30	1.16	1.121
3mm	21.071	2.041	4.836	0.422	3.752	0.365	5.62	5.59	1.29	1.15	1.557
atax	0.025	0.002	0.019	0.002	0.009	0.002	2.73	1.30	2.06	1.18	0.282
bicg	0.022	0.002	0.019	0.002	0.009	0.002	2.41	1.13	2.09	1.13	0.285
cholesky	11.951	57.895	7.714	1.006	7.500	0.985	1.59	58.77	1.03	1.02	0.403
correlation	9.564	1.537	1.960	0.337	2.456	0.413	3.89	3.72	0.80	0.82	1.17
covariance	10.510	1.806	2.456	0.434	2.836	0.539	3.71	3.35	0.87	0.81	0.939
doitgen	3.654	1.088	0.917	0.924	0.858	0.917	4.26	1.19	1.07	1.01	0.333
durbin	0.019	0.050	0.014	0.020	0.014	0.020	1.35	2.56	1.00	1.00	-
fdtd-2d	24.486	24.687	19.177	1.861	19.277	2.203	1.27	11.21	0.99	0.84	0.981
floyd-warshall	189.56	190.17	271.325	30.689	272.05	30.268	0.70	6.28	1.00	1.01	0.13
gemm	12.549	0.984	2.792	0.204	2.101	0.174	5.97	5.67	1.33	1.17	0.548
gemver	0.209	0.018	0.133	0.012	0.048	0.011	4.32	1.58	2.74	1.06	0.521
gesummv	0.054	0.005	0.049	0.004	0.019	0.004	2.81	1.34	2.53	1.18	0.245
gramschmidt	152.62	39.635	12.804	2.015	10.780	1.988	14.16	19.93	1.19	1.01	0.65
heat-3d	119.02	946.22	25.068	7.142	32.011	6.728	3.72	140.6	0.78	1.06	2.389
jacobi-1d	0.009	0.012	0.010	0.002	0.010	0.001	0.91	8.54	0.97	1.27	0.097
jacobi-2d	26.749	28.541	30.042	2.592	29.832	2.411	0.90	11.84	1.01	1.07	0.566
lu	20.551	58.218	16.767	1.729	14.501	1.536	1.42	37.89	1.16	1.13	0.455
mvt	0.107	0.010	0.136	0.011	0.099	0.008	1.09	1.15	1.38	1.30	0.186
seidel-2d	172.80	174.04	173.644	17.177	201.16	19.989	0.86	8.71	0.86	0.86	0.374
symm	20.262	14.899	55.523	55.525	55.828	55.828	0.36	0.27	0.99	0.99	-
syr2k	10.548	1.480	9.129	1.257	9.389	1.343	1.12	1.10	0.97	0.94	0.656
syrk	6.776	0.956	5.037	0.694	4.842	0.674	1.40	1.42	1.04	1.03	0.487
trisolv	0.065	0.760	0.041	0.004	0.040	0.004	1.64	187.3	1.03	1.02	0.101
trmm	5.419	0.485	1.401	0.146	1.250	0.134	4.34	3.61	1.12	1.08	0.423
geomean							**2.14**	**5.11**	**1.17**	**1.04**	

Table 7: Execution time (s) and tile size selection time (ms) for digital signal processing benchmarks.

Benchmark	Scipy		Matlab		PolyMage-fft		PolyMage-opt		PolyMage-opt speedup			Tile size selection
	1	16	1	16	1	16	1	16	SciPy	Matlab	PolyMage-fft	time (ms)
Vuvuzela	5.848	8.972	7.633	0.958	13.92	1.123	10.055	0.886	10.13	1.08	1.27	34.51
Unwanted Spectral	0.225	0.294	0.289	0.097	0.234	0.022	0.206	0.021	13.86	4.57	1.04	15.92
geomean									**11.85**	**2.22**	**1.15**	

5.4 Compilation Time Overhead of the Tile Size Selection Model

In this section, we provide the details on time taken by the tile size selection model described in Section 3. Column 13 in Table 4 and Table 7 provide the time taken to compute tile sizes for the linear algebra benchmarks and benchmarks from the DSP domain respectively. It is evident from table, that the time taken for tile size selection is very small. The vuvuzela filter takes maximum time since tile sizes had to be found for 32 permutable bands, the largest among the benchmarks that we had considered. Note that, the compile times shown in Table 4, Table 6 and Table 7, illustrate that the time taken by the tile size selection model is significantly smaller than the execution time of the benchmarks even for smaller datasets. In symm and durbin, the loop nest is not tiled in case of Pluto and hence no time is spent in the tile size selection model (refer Table 6, last column). Note that, the performance gain due to the model is significantly higher, especially in case of *PolyMage-tss*, which motivates us to incorporate the proposed tile size selection model in an infrastructure like MLIR, which provides support for parametric tiling, and the actual tile sizes can be determined at runtime.

To summarize our experiments, our optimizations described in Section 3 (PolyMage-tss) provides a performance improvement of 1.2× over default tile sizes for linear algebra benchmarks. For the entire PolyBench suite we observe a mean speedup of 1.04× over Pluto. For DSP benchmarks, we observe that our approach performed better than Matlab and Intel's SciPy by 2.2× and 11× respectively. The variance in data was very small. The highest variance was 2.04E-5 for extralarge dataset and 1.24E-9 for the medium dataset for syrk and syr2k benchmarks respectively. We also observed that the model found the tile sizes almost instantaneously, making it suitable for incorporation in a general-purpose compiler infrastructure like MLIR.

6 RELATED WORK

Tile size selection models have been proposed in the context of general-purpose and domain-specific compilation for incorporation in both research and production compilers. Sarkar and Megiddo [27] compute tile sizes in IBM XL Fortran compiler using a cost function based on the distinct cache lines and pages in TLB accessed in the tile for a two dimensional loop nest. Then, by solving an optimization problem, tile sizes are found such that the cost is minimized. For higher dimensions, they follow an iterative approach, in which, tiles sizes for the outer dimensions are found by an exploratory search, while those for the innermost two dimensions are found using the optimization problem, thereby resorting to a mixture of auto-tuning and analytical modeling for higher dimensional loop nests. Renganarayanan and Rajopadhye [26] formalize various tile size selection models proposed in literature using posynomials. They formulate a non-linear Integer Geometric Programming [5] with the objective of minimizing the memory footprint of a tile. These complex IGP formulations can incur significantly large compile time costs, thereby making these models unsuitable for adoption in general-purpose compiler infrastructure. As a part of the future work, we plan to avoid the IGP formulations proposed in the literature by the dimensional reuse metric described in Section 3.

Polyhedral auto-transformation frameworks like Pluto and PPCG perform loop tiling efficiently. However, they do not have a tile size selection model and typically rely on auto-tuning in a space of tile sizes. A number of past works have thus evaluated and compared with Pluto and PPCG with their respective default tile sizes. The approaches of [19, 29, 38] aim at finding tile sizes for use in a polyhedral compiler framework. However, they have the following limitations: (i) assume a single level of rectangular tiling [19], (ii) find equal tile sizes for all dimensions [38], (iii) provide a bound on the search space of tiles sizes [29], and rely on auto-tuning in a smaller search space. However, tuning in this smaller space can be significantly costly, even in simple examples like matrix multiplication. Feld et. al, [8] propose a simple tile size selection model based on the number of data elements in a tile that fit in multiple levels of the cache hierarchy. They do not consider any reuse, and hence, choosing a larger tile size along a dimension that does not have any reuse might not be profitable. On the other hand, our approach computes tile sizes that are proportional to reuse along dimensions, enables better vectorization and prefetching, ensuring that each thread has sufficient computation. Our approach can also be extended to tiling at multiple levels using an approach similar to that of Feld et. al, [8]. Table 8 summarizes the various works available on tile size selection models.

Table 8: Summary of tile size selection models.

Work	Method used to find tile size	Tool Available?
Sarkar and Megiddo [27]	Cost function + Auto tuning	No
Renganarayanan and Rajopadhye [26]	Cost function (high compile time)	No
Mehta et. al. [19]	Auto tuning	No
Shirako et. al. [29]	Auto tuning	No
Yuki et.al. [38]	Auto tuning (finds equal tile sizes)	No
Feld et. al. [8]	Cost function (reuse not considered)	No
PolyMage [13]	Cost function (limited to stencils)	Yes
Pluto [22]	Default tile size of 32	Yes
PPCG [25]	Default tile size of 16	Yes

Domain specific tile size selection approaches: Vendor specific libraries like OpenBLAS, MKL, Eigen [7, 20, 35] rely on hand optimized implementations for high performance, where tile sizes are tuned by experts for various problem sizes. Attempts to automate these optimizations include LAPACK [1], PHiPAC [3] and ATLAS [36]. They iteratively tune for performance by varying, block sizes, loop orders and also use runtime feedback mechanism for auto-tuning. All these libraries provide excellent performance for large problem sizes but their performance can be sub-optimal for small matrices, as shown by the results in the work of Hall et. al [28]. Moreover, highly optimized libraries may not be available for all computation patterns in the application, and hence, using an optimizing compiler along side optimized libraries can provide significant compiler improvements, as demonstrated by our experiments in Section 5.3.2.

Problem-specific tile size selection models have been widely studied to optimize matrix-matrix multiplications [9, 18] and convolutions [17] as these computations are ubiquitous in machine learning applications. These models can be used to generate optimized *microkernels*, which can then be composed together to obtain high-performant primitives that perform on par with highly optimized vendor specific libraries [33, 40]. The scope of the proposed tile size selection model is more broader, and is targeted for incorporation in a generic compiler infrastructure like GCC, LLVM and MLIR.

Several domain specific languages and compilers [2, 6, 10, 12, 16, 30, 31, 39] have been proposed in literature with the aim of optimizing dense linear algebra computations. FLAME [6, 10, 39] requires the programmer to decompose complex matrix operations into operations on smaller blocks, which are then mapped to optimized library routines. The associated runtime addresses issue of load balancing using task-parallelism. However, the programmer must find efficient block sizes in order to fully utilize the benefits of the underlying runtime. Domain-specific compilers like Tensor Comprehensions [34], Ansor [41] rely on auto-tuning for finding good tile sizes by using a genetic algorithm. This auto-tuning step takes a considerable amount of time even when restricted to specific applications in deep learning.

The tile size selection model of Jangda et al. [13], is the state-of-the-art for image processing pipelines. The model assumes that there exists one-to-one correspondence between iterations and data accessed within a group i.e., the product of tile sizes along each dimension is equal to the number of distinct memory accesses within a tile (tile volume). This assumption of one-to-one correspondence between iterations and data accessed holds true for image processing pipelines but does not hold true for simple linear algebra computations like matrix-matrix multiplication. Moreover, while considering image processing applications, we demonstrated that our tile size selection model finds very similar tile sizes as their approach (refer Section 5.3.3), proving the effectiveness of our model in finding good tile sizes without significant compile time overheads.

7 CONCLUSIONS

In this paper, we proposed a fast, generic and robust tile size selection model which can provide tile sizes for affine loop nests with arbitrary affine accesses. The model considered various aspects like data reuse, cache capacity, vectorization, and load balancing issues without introducing significant compile time overhead. We implemented the model in PolyMage and Pluto. We further enhanced PolyMage to support mapping to optimized libraries whenever profitable. Our approach obtained good scalable speedups on benchmarks from different domains for various problem sizes, demonstrating the generality of the tile size selection model. We observed a mean speedup of 11.8× over Intel's Scipy and 2.2× over MATLAB for filters from the DSP domain. We observed a mean speedup of 1.04× (maximum speedup of 1.3×) over Pluto for the entire PolyBench suite. For the linear algebra benchmarks in PolyBench, we observed a mean performance improvement of 1.24× (maximum speedup of 3.65×) over PolyMage. These performance gains along with a negligible compile time overhead motivate the incorporation of the proposed tile size selection model in general-purpose compiler infrastructures like those of LLVM, GCC or MLIR.

ACKNOWLEDGEMENTS

We would like to thank the Science and Engineering Research Board (SERB), India for a grant under the EMR program (EMR/2016/008015) which has supported this research work in part. We are also grateful to Google Research for a grant through a Google faculty research award in 2015.

REFERENCES

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A Portable Linear Algebra Library for High-performance Computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* (New York, New York, USA) *(Supercomputing '90)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 2–11. http://dl.acm.org/citation.cfm?id=110382.110385

[2] Denis Barthou, Paul Feautrier, and Xavier Redon. 2002. On the Equivalence of Two Systems of Affine Recurrence Equations (Research Note). (01 2002), 309–313.

[3] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 2014. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (Munich, Germany). ACM, New York, NY, USA, 253–260. https://doi.org/10.1145/2591635.2667174

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN conference on Programming Languages Design and Implementation* (Tucson, AZ, USA). 101–113. https://doi.org/10.1145/1375581.1375595

[5] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization.* Cambridge University Press.

[6] Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. 2007. Supermatrix Out-of-order Scheduling of Matrix Operations for SMP and Multi-core Architectures. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (San Diego, California, USA) *(SPAA '07)*. ACM, New York, NY, USA, 116–125. https://doi.org/10.1145/1248377.1248397

[7] Eigen [n.d.]. A C++ template library for linear algebra. http://eigen.tuxfamily.org/.

[8] Dustin Feld, Thomas Soddemann, Michael Jünger, and Sven Mallach. 2013. Facilitate SIMD-Code-Generation in the polyhedral model by hardware-aware automatic code-transformation. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques.* Citeseer, 45–54.

[9] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[10] John A. Gunnels and Robert A. van de Geijn. 2000. *Formal Linear Algebra Methods Environment (FLAME) Overview.* Technical Report. Austin, TX, USA.

[11] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis.* 981–991. https://doi.org/10.1109/SC.2016.83

[12] Guillaume Iooss. 2016. *Detection of linear algebra operations in polyhedral programs.* Theses. Université de Lyon. https://tel.archives-ouvertes.fr/tel-01370553

[13] Abhinav Jangda and Uday Bondhugula. 2018. An Effective Fusion and Tile Size Model for Optimizing Image Processing Pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18).* ACM, New York, NY, USA, 261–275. https://doi.org/10.1145/3178487.3178507

[14] Martin Kong and Louis-Noël Pouchet. 2019. Model-Driven Transformations for Multi- and Many-Core CPUs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019).* 469âĂŞ484.

[15] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL]

[16] lgen-spiral [n.d.]. LGen: A Basic Linear Algebra Compiler. http://www.spiral.net/software/lgen.html.

[17] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. 2021. Analytical Characterization and Design Space Exploration for Optimization of CNNs. arXiv:2101.09808 [cs.LG]

[18] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (Aug. 2016), 18 pages. https://doi.org/10.1145/2925987

[19] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. 2013. Tile Size Selection Revisited. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4, Article 35 (Dec. 2013), 27 pages. https://doi.org/10.1145/2541228.2555292

[20] MKL [n.d.]. Intel math kernel library (MKL). http://software.intel.com/en-us/intel-mkl.

[21] Ravi Teja Mullapudi, Vinay Vasista, and Uday. Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[22] Pluto 2008. Pluto: An automatic polyhedral parallelizer and locality optimzer. https://github.com/bondhugula/pluto.

[23] Polybench v4.2 2016. Polybench suite. http://polybench.sourceforge.net.

[24] PolyMage project, Apache 2.0 license 2016. PolyMage. https://bitbucket.org/udayb/polymage commit 0ff0b46456605a5579db09c6ef98cb247dd2131d, Dec 16, 2016.

[25] PPCG 2013. Polyhedral Parallel Code Generation. https://github.com/Meinersbur/ppcg.

[26] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. 2008. Positivity, Posynomials and Tile Size Selection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Austin, Texas) *(SC '08).* IEEE Press, Article 55, 12 pages.

[27] V. Sarkar and N. Megiddo. 2000. An Analytical Model for Loop Tiling and Its Solution. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '00).* IEEE Computer Society, USA, 146âĂŞ153.

[28] Jaewook Shin, Mary Hall, Jacqueline Chame, Chun Chen, and Paul D. Hovland. 2009. Autotuning and Specialization: Speeding up Matrix Multiply for Small Matrices with Compiler Technology. (01 2009).

[29] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. 2012. Analytical Bounds for Optimal Tile Size Selection. In *Proceedings of the 21st International Conference on Compiler Construction* (Tallinn, Estonia) *(CC'12).* Springer-Verlag, Berlin, Heidelberg, 101–121. https://doi.org/10.1007/978-3-642-28652-0_6

[30] J. G. Siek, I. Karlin, and E. R. Jessup. 2008. Build to order linear algebra kernels. In *2008 IEEE International Symposium on Parallel and Distributed Processing.* 1–8. https://doi.org/10.1109/IPDPS.2008.4536183

[31] Daniele G. Spampinato and Markus Püschel. 2014. A Basic Linear Algebra Compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14).* ACM, New York, NY, USA, Article 23, 10 pages. https://doi.org/10.1145/2544137.2544155

[32] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2014. A Framework for Enhancing Data Reuse via Associative Reordering. (June 2014).

[33] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Bharat Kaul, Gagandeep Goyal, and Ramakrishna Upadrasta. 2021. PolyDL: Polyhedral Optimizations for Creation of High-Performance DL Primitives. *ACM Trans. Archit. Code Optim.* 18, 1, Article 11 (Jan. 2021), 27 pages. https://doi.org/10.1145/3433103

[34] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[35] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13).* ACM, New York, NY, USA, Article 25, 12 pages. https://doi.org/10.1145/2503210.2503219

[36] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2000. Automated Empirical Optimization of Software and the ATLAS Project. *PARALLEL COMPUTING* 27 (2000), 2001.

[37] M. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation.* 30–44.

[38] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. 2010. Automatic Creation of Tile Size Selection Models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10).* ACM, New York, NY, USA, 190–199. https://doi.org/10.1145/1772954.1772982

[39] F. G. V. Zee, E. Chan, R. A. v. d. Geijn, E. S. Quintana-OrtĂŋ, and G. Quintana-OrtĂŋ. 2009. The libflame Library for Dense Matrix Computations. *Computing in Science Engineering* 11, 6 (Nov 2009), 56–63. https://doi.org/10.1109/MCSE.2009.207

[40] F. V. Zee and R. Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41 (2015), 14:1–14:33.

[41] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* https://www.usenix.org/conference/osdi20/presentation/zheng